

Міністерство освіти і науки України  
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА  
АКАДЕМІЯ»

Кафедра інформатики факультету інформатики



**Тестування у Haskell**

**Текстова частина до курсової роботи  
за спеціальністю «Комп'ютерні науки» - 122**

**Керівник курсової роботи**

Кандидат фізико-математичних наук,

доцент

Проценко В.С.

---

(підпис)

“ \_\_\_\_ ” \_\_\_\_\_ 2020 р.

**Виконала студентка КН-3:**

Яськова Д.В.

“ \_\_\_\_ ” \_\_\_\_\_ 2020 р.

Міністерство освіти і науки України  
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЇВО-МОГИЛЯНСЬКА  
АКАДЕМІЯ»  
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ  
Кандидат фізико-математичних наук,  
доцент  
Проценко В.С.

\_\_\_\_\_  
(підпис)

„ \_\_\_\_ ” \_\_\_\_\_ 2019р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ  
на курсову роботу  
студентці Яськовій Дарині Вадимівні  
факультету інформатики 3 курсу бакалаврської програми  
**ТЕМА: Тестування у Haskell**

Зміст ТЧ до курсової роботи:

Анотація

Вступ

Розділ 1. Налаштування проекту для тестування

Розділ 2. Основні принципи тестування та їх приклади на Haskell

Висновок

Список використаної літератури

Дата видачі „ \_\_\_\_ ” \_\_\_\_\_ 2019 р.

Керівник \_\_\_\_\_  
(підпис)

Завдання отримав \_\_\_\_\_  
(підпис)

## Календарний план виконання роботи

№	Назва етапу	Термін виконання	Примітка
1.	Отримання завдання на курсову роботу	10.10.2019	
2.	Огляд технічної літератури за темою роботи	05.11.2019	
3.	Написання основного застосунку для тестування	20.11.2019	
4.	Вивчення основних принципів тестування	15.12.2019	
5.	Пошук бібліотек написаних на Haskell	05.01.2020	
7.	Визначення структури програми	15.01.2020	
8.	Написання першої частини курсової роботи	20.02.2020	
9.	Перегляд змісту роботи з керівником	25.03.2020	
11	Написання другої частини курсової роботи	15.04.2020	
12	Внесення змін до роботи	04.05.2020	
17	Створення презентації	09.05.2020	
18	Захист роботи	18.05.2020	

## Зміст

<b>Анотація .....</b>	<b>5</b>
<b>Вступ .....</b>	<b>6</b>
<b>Розділ 1. Налаштування проекту для тестування .....</b>	<b>8</b>
1.1 Використання інструменту Stack для організації проекту .....	8
1.1.1 Загальні відомості .....	8
1.1.2 Приклад роботи з інструментом Stack .....	9
1.2 Підключення бібліотек .....	13
1.3 Фреймворк Hspec для організації тестів .....	16
1.3.1 Загальні відомості .....	16
1.3.2 Огляд роботи з фреймворком .....	18
<b>Розділ 2. Основні принципи тестування та їх приклади на Haskell .....</b>	<b>22</b>
2.1 Unit testing .....	22
2.1.1 Загальні відомості .....	22
2.1.2 Бібліотека HUnit .....	23
2.1.3. Приклад написання Unit-тестів з використанням HUnit .....	25
2.2 Property testing .....	28
2.2.1 Загальні відомості .....	28
2.2.2 Бібліотека QuickCheck .....	31
2.2.3 Приклад написання тестів з використанням бібліотеки QuickCheck .....	33
2.2.4 Бібліотека SmallCheck .....	36
2.2.5 Приклад написання тестів з використанням бібліотеки SmallCheck .....	38
2.3 Mocking .....	40
2.3.1 Загальні відомості .....	40
2.3.2 Приклад написання тестів .....	42
<b>Висновок .....</b>	<b>47</b>
<b>Список використаної літератури .....</b>	<b>48</b>

## **Анотація**

У цій роботі детально розглядається процес тестування на мові функціонального програмування Haskell як частина перевірки якості розробленого застосунку, в ролі якого було обрано інтерпретатор. Основною метою було проаналізувати принцип роботи існуючих бібліотек та продемонструвати їх на власному прикладі.

## **Вступ**

### **Актуальність теми**

На сьогоднішній день тестування є невід'ємною частиною під час розробки будь-якого застосунку. Це процес, який перевіряє чи дійсно код працює коректно та робить те, що задумано і водночас нічого зайвого. Метою тестування є перевірка чи однакова реальна поведінки програми із очікуваним результатом її виконання на певному наборі тестів.

Тестування забезпечує:

- знаходження помилок при написанні коду;
- відповідність ТЗ та початковим вимогам;
- перевірку на витрачений час та пам'ять;
- малу вірогідність зруйнувати логіку функцій при їх зміні;
- позбавлення користувачів від неприємних сюрпризів.

Тестування є гарною можливістю уникнути багатьох проблем у майбутньому. Зазвичай, воно є гарним показником якості розробленого продукту. Також, іноді тестування слугує гарною технічною документацією як для старих, так і для нових розробників.

Під час роботи розробники часто стикаються із задачею, де необхідно змінити чужий код і погано розуміючи логіку вже написаного, навіть невеликі зміни можуть принести багато проблем. У такому випадку за допомогою тестів легко контролювати будь-які зміни, аби уникнути зайвих логічних помилок.

Якщо розглядати тестування на мові Haskell, то на просторах Інтернету не так багато інформації про нього, у порівнянні із такими мовами

програмування, як Java, C++, C# та інші. Тим більше для кожної бібліотеки необхідно продивитись мінімум декілька ресурсів, аби зрозуміти принцип її роботи, а таких бібліотек багато. Тому виникає потреба у одному великому ресурсу, де буде зібрана інформація про всі основні методи тестування на Haskell, як їх приєднати до проекту та як правильно використати. Із цього випливає, що тема курсової роботи є актуальною.

### **Мета дослідження**

Дослідити основні методи тестування за допомогою різних бібліотек у Haskell та об'єднати їх у єдиний ресурс.

### **Постановка задачі**

1. вивчення основних принципів тестування;
2. ознайомлення із бібліотеками у мові Haskell;
3. вибір основного застосунку та його створення;
4. тестування, використовуючи вивчені бібліотеки та принципи, на вже готовому застосунку.

## Розділ 1. Налаштування проекту для тестування

### 1.1 Використання інструменту Stack для організації проекту

#### 1.1.1 Загальні відомості

Stack - це крос-платформна програма для розробки проектів у Haskell. Через гарну документацію і зручний інтерфейс вона буде хорошим вибором як для початківців, так і для розробників із великим досвідом. Її унікальністю є те, що програм із схожим функціоналом дуже мало для Haskell.

Переваги:

- Допомогає швидко завантажити необхідні пакети;
- Збирає проект;
- Створює файл для тестування, який надалі запускається однією командою;
- Автоматично встановлює систему збору Cabal, що займається організацією бібліотек;
- Надає доступ до Hackage Package Repository – архів, у якому зібрано велику кількість бібліотек.

Основні команди:

- `stack new <my-project>` – команда, що створює каталог із назвою проекту `my-project`. Проект створюється за певним шаблоном, якщо його не передати додатковим аргументом, то `stack` буде використовувати шаблон за замовчуванням;
- `stack build` – команда, котра будує проект;
- `stack exec my-project-exe` – команда, котра запускає створений після компіляції виконуваний файл;



- `stack test` – команда, котра запускає процес тестування.

### 1.1.2 Приклад роботи з інструментом Stack

Для початку необхідно створити проект у консолі за допомогою команди `stack new <my-project>`:

```
Dasha@DESKTOP-UBGP5LU MINGW64 ~/Desktop/testing  
$ stack new cursova
```

Після виконання команди, створиться каталог із наступними файлами:

```
|  
|  .gitignore  
|  ChangeLog.md  
|  cursova.cabal  
|  LICENSE  
|  package.yaml  
|  README.md  
|  Setup.hs  
|  stack.yaml  
|  
+---app  
|      Main.hs  
|  
+---src  
|      Lib.hs  
|  
\---test  
      Spec.hs
```

Розглянемо детальніше дерево файлів проекту[1]. `cursova.cabal` є конфігураційним файлом. У ньому зберігаються метадані пов'язані із проектом. Також зберігається така базова інформація, як ім'я проекту, його опис, версія тощо:

```
name:          cursova
version:       0.1.0.0
description:   Please see the README on GitHub at <https://github.com/githubuser/cursova#readme>
homepage:     https://github.com/githubuser/cursova#readme
bug-reports:  https://github.com/githubuser/cursova/issues
author:       Author name here
maintainer:   example@example.com
copyright:   2020 Author name here
license:     BSD3
license-file: LICENSE
build-type:  Simple
extra-source-files:
  README.md
  ChangeLog.md
```

У секції `library` знаходиться інформація про бібліотеки, які використовує проект, та каталог, у якому зберігаються бібліотечні файли:

```
library
  exposed-modules:
    | Lib
  other-modules:
    | Paths_cursova
  hs-source-dirs:
    | src
```

У секції `executables` зберігається інформація про файли, які будуть використані для побудови виконуваного файлу проекту, та каталог, у якому вони знаходяться:

```
executable cursova-exe
  main-is: Main.hs
  other-modules:
    Paths_cursova
  hs-source-dirs:
    app
```

У секції `test-suite` зберігається інформація про файли, пов'язані із тестуванням, та каталог, у якому вони зберігаються:

```
test-suite cursova-test
  type: exitcode-stdio-1.0
  main-is: Spec.hs
  other-modules:
    Paths_cursova
  hs-source-dirs:
    test
```

Підкаталог `app` призначений, як було вище розглянуто, для виконуваних модулів, підкаталог `src` – для бібліотечних модулів, а підкаталог `test` – для тестів. Файли `Main.hs`, `Lib.hs` та `Spec.hs` генеруються автоматично при створенні проекту.

Розібравшись із структурою файлів, перейдемо до написання коду. Додаємо функції для роботи із інтерпретатором у файл `Lib.hs`. У файлі `Main.hs` формуємо головну функцію, котра буде виглядати наступним чином:

```

module Main where
import Lib

main :: IO ()
main = do
    name <- getLine
    x <- parseFile name
    putStrLn $ show $ x

```

За допомогою вбудованої функції `getLine` зчитується назва файлу з клавіатури та передається до функції `parseFile`, котра знаходиться у модулі `Lib`:

```

parseFile :: String -> IO ProgramVariables
parseFile file = do
    input_info <- readFile file
    case parseCode input_info of
        Left e  -> error $ show e
        Right r -> return (evaluateProgram r)

```

Функція `parseFile` аналізує файл із кодом, виконує його та повертає у результаті оновлений стан програми, який виводиться у вихідний потік. Під станом програми йде мова про список змінних зі значеннями.

Далі треба перевірити чи працює написана програма. Щоб побудувати проект, треба виконати команду `stack build`. Перший раз виконання команди може зайняти до декількох хвилин. Далі виконуємо команду `stack exec <my-project-exe>`, щоб запустити програму:

```
Dasha@DESKTOP-UBGP5LU MINGW64 ~/Desktop/Haskell/cursova (master)
$ stack exec cursova-exe
```

Вводимо назву (абсолютний шлях) файлу і отримуємо результат:

```
Dasha@DESKTOP-UBGP5LU MINGW64 ~/Desktop/Haskell/cursova (master)
$ stack exec cursova-exe
F:/power.txt
[("a",I 317),("b",I 17)]
```

Файл `power.txt` містить код, у якому оголошується декілька змінних та цикл `while`, котрий змінює значення змінних та виконується доки деяке арифметичне твердження істинне:

```
{
  int a; int b; a := 317; b := 0;
  { bool c; c:=true; while(c) {b++; c:= a >= b*b} };
  b := b-1
}
```

Рисунок 1.1 Вигляд файлу `power.txt`

## 1.2 Підключення бібліотек

Налаштовуючи проект, підключення бібліотек виявилось найбільшою проблемою, тому є доречним розібрати детально це питання.

В офіційній документації рекомендується додавати бібліотеки через зміну файлу `package.yaml`. Цей файл генерується під час створення

проекту інструментом Stack. У документації написано, що цей файл підтримує формат `hspac` для пакетів – альтернативу `Cabal` формату. `Hspac` підтримує синтаксис файлу `.yaml` і може згенерувати з нього файл із розширенням `.cabal`. Фактично, цей формат є зручною обгорткою для роботи із форматом `Cabal`.<sup>[2]</sup>

Файл зберігає ім'я проекту, його версію, загальну інформацію проекту, налаштування для тестування та залежності, тобто підключені допоміжні бібліотеки.

Розглянемо на конкретному прикладі додавання бібліотеки `QuickCheck`. Перш за все необхідно в інтернеті перевірити точну назву бібліотеки, адже помилившись навіть у реєстрі буде помилка. Так наприклад бібліотека `QuickCheck` містить дві великі літери у той час, коли `smallcheck` – жодної. Загалом використовується `kebab-case` – стиль написання слів, у якому пробіли замінюються на дефіс. Прикладом є `hspec-contrib`, `quickcheck-instances`, `monad-mock` тощо.

Наступним кроком додаємо бібліотеку до розділу `dependencies` у файлі `package.yaml`:

```

name:          cursova
version:       0.1.0.0
github:        "githubuser/cursova"
license:       BSD3
author:        "Author name here"
maintainer:    "example@example.com"
copyright:     "2020 Author name here"

extra-source-files:
- README.md
- ChangeLog.md

# Metadata used when publishing your package
# synopsis:      Short description of your package
# category:      Web

# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description:    Please see the README on GitHub at <https://github.com/githubuser/cursova#readme>

dependencies:
- base >= 4.7 && < 5
- QuickCheck

```

Щоб перевірити чи вдало додалась бібліотека, необхідно виконати в консолі команду `stack build`:

```

$ stack build
cursova-0.1.0.0: unregistering (local file changes: src\Lib.hs)
cursova> build (lib + exe)
Preprocessing library for cursova-0.1.0.0..
Building library for cursova-0.1.0.0..
[3 of 3] Compiling Lib
Preprocessing executable 'cursova-exe' for cursova-0.1.0.0..
Building executable 'cursova-exe' for cursova-0.1.0.0..
cursova> copy/register
Installing library in C:\Users\Dasha\Desktop\Haskell\cursova\.stack-work\install\fd5e69c6\lib\x86_64-windows-ghc-8.6.5\cursova-0.1.0.0-5He1btkKOgjAViUQj7B6m3
Installing executable cursova-exe in C:\Users\Dasha\Desktop\Haskell\cursova\.stack-work\install\fd5e69c6\bin
Registering library for cursova-0.1.0.0..

```

Якщо після виконання команди ніяких помилок не було виявлено, можна додавати модуль `Test.QuickCheck` до файлу `Spec.hs`.

## 1.3 Фреймворк Hspec для організації тестів

### 1.3.1 Загальні відомості

Hspec - це фреймворк, який було створено для спрощення процесу тестування у Haskell. Зручність полягає в автоматизації тестів та їх структуризації.

Його перевагами є:

- взаємодія з популярними бібліотеками для тестування (надалі будуть розглянуті `SmallCheck`, `QuickCheck`, `HUnit`);
- структуризація тестів за допомогою ділення їх на блоки та короткого опису кожного;
- паралельне виконання тестів.

Функції, що надає фреймворк Hspec можна поділити на декілька блоків. А саме[3]:

#### 1. Блок для опису тестів та їх структуризації.

До цього блоку можна віднести такі функції, як `describe`, `context` та `it`. За допомогою цих функцій можна організувати тести у логічні групи та зробити опис кожного тесту без використання додаткових коментарів у коді.

```
it :: (HasCallStack, Example a) => String -> a -> SpecWith (Arg a)
```

Функція `it` створює специфікацію тесту. Специфікація складається із опису тесту та прикладу поведінки тесту.



```
describe :: HasCallStack => String -> SpecWith a -> SpecWith a
```

Функція `describe` об'єднує список специфікацій у одну велику специфікацію. Ця функція знаходиться на найвищому рівні вкладеності.

```
context :: HasCallStack => String -> SpecWith a -> SpecWith a
```

Функція `context` є псевдонімом для функції `describe`. Зазвичай її використовують для додаткового опису специфікації зі словами “коли” або “з”.

Також до цього блоку можна віднести функції `pending` та `pendingWith`.

```
pending :: HasCallStack => Expectation
```

Функція `pending` дозволяє віднести специфікацію до стану очікування. Це дозволяє оголосити тест, але не писати одразу його реалізацію, а відкласти на потім.

```
pendingWith :: HasCallStack => String -> Expectation
```

Функція `pendingWith` працює аналогічно до `pending`, але додатково приймає стрічку із коментарем, чому специфікація віднесена до стану очікування.

## 2. Блок для перевірки результатів тестування.

```
shouldBe :: (HasCallStack, Show a, Eq a) => a -> a -> Expectation
```

Функція `shouldBe` стверджує, що очікуваний результат є еквівалентним до отриманого.

```
shouldSatisfy :: (HasCallStack, Show a) => a -> (a -> Bool) -> Expectation
```

Функція `shouldSatisfy` стверджує, що предикат є успішним для деякого значення. Ця функція є схожою до функції `assertBool` з `HUnit`.

## 3. Блок для виконання тестів

```
hspec :: Spec -> IO ()
```

Функція `hspec` є головною функцією, яка відповідає за виконання написаних специфікацій. Результат виконання виводиться у стандартний потік виводу.

### 1.3.2 Огляд роботи з фреймворком

Щоб розпочати роботу, треба додати пакет `hspec` до залежностей та імпортувати модулі `Test.Hspec`, `Lib` у файлі `Spec.hs`.

Розглянемо на конкретних прикладах декілька тестів. Необхідно протестувати дві функції з файлу `Lib.hs` – `getValFromState` та `updateValFromState`:

```
getValFromState :: ProgramVariables -> String -> Value
getValFromState state variable = fromJust $ lookup variable state

updateValFromState :: ProgramVariables -> String -> Value -> ProgramVariables
updateValFromState [] _ _ = []
updateValFromState (s:state) variable value
  | fst s == variable = (fst s, value):state
  | otherwise = s:updateValFromState state variable value
```

Перша функція приймає масив із кортежами, де перше значення – ім'я змінної, а друге – значення змінної. Другим аргументом функція приймає ім'я змінної, значення якої треба знайти. Принцип роботи полягає у тому, що функція пробіжить по масиву, порівнюючи надане ім'я змінної зі значеннями у масиві та у результаті поверне її значення, якщо знайде, або помилку, якщо такої змінної у масиві не буде.

Друга функція працює аналогічно до першої. Першим аргументом функція приймає масив змінних, другим – ім'я змінної, значення якої треба оновити, а третім аргументом - нове значення. В результаті функція повертає оновлений масив змінних.

Зараз і надалі всі тести додаємо до файлу `Spec.hs`:

```

main :: IO ()
main = hspec $ do
  describe "Lib.getValFromState" $ do
    1 тест it "returns value of the second variable in list" $ do
      (getValFromState [("y", (I 5)), ("x", (I 6)), ("V", (B True))] "x")
      `shouldBe` ((I 6) :: Value)
    2 тест it "returns value of the first variable in list" $ do
      (getValFromState [("y", (I 5)), ("x", (I 6)), ("V", (B True))] "y")
      `shouldSatisfy` (== (I 5))
    3 тест describe "Lib.updateValFromState" $ do
      it "" $ pending

```

Тестування першої функції відбувається за допомогою двох тестів. Функція `describe` описує назву функції, що тестується – “`Lib.getValFromState`”. Далі функція `it` описує окремо для кожного тесту бажану поведінку функції. У першому тесті за допомогою функції `shouldBe` порівнюється очікуваний результат із тим, що повертає функція `getValFromState`. У другому тесті використовується функція `shouldSatisfy`, яка працює за схожим принципом до `shouldBe`, але замість конкретного значення приймає предикат.

На прикладі тестування другої функції демонструється використання функції `pending`, за допомогою якої можна відкласти реалізацію тесту на певний час.

Щоб запустити тести, необхідно в консолі виконати команду `stack test`. В консолі виведено тести із їх описом та результат виконання – 3 тести виконались успішно, 1 з яких знаходиться у стані очікування:

```
$ stack test
cursova> test (suite: cursova-test)

Lib.getValFromState
  returns value of the second variable in list
  returns value of the first variable in list
Lib.updateValFromState
Main[145:7]
  # PENDING: No reason given

Finished in 0.0007 seconds
3 examples, 0 failures, 1 pending

cursova> Test suite cursova-test passed
```

Також, використовуючи Hspec, можливо інтегруватися з бібліотеками для тестування. Детально цей процес описано у другому розділі для кожної з бібліотек.

## Розділ 2. Основні принципи тестування та їх приклади на Haskell

### 2.1 Unit testing

#### 2.1.1 Загальні відомості

Unit-тестування – це процес, що дозволяє перевірити коректність роботи окремих модулів. Головна ідея полягає у тому, що для кожної функції існує тест для її перевірки.

Перш за все, тести мають бути читабельні і, як і основний код, відповідати гарному тону проектування. Для цього їм необхідно дотримуватись наступних правил FIRST[4]:

- *Fast* - тести повинні бути швидкими. Це необхідно для швидкого запуску, адже їх доведеться запускати велику кількість разів у майбутньому;
- *Independent* – тести повинні бути незалежними та запускатися у будь-якому порядку. Це є необхідним, бо якщо тести залежні між собою, то помилка одного призводить до помилки інших, що руйнує усі принципи тестування;
- *Repeatable* – тести мають такими, щоб їх було легко запускати багато разів. Вони мають запускатись у будь-якому середовищі, незалежно від того чи це локальне середовище, чи на production;
- *Self-validating* – тести повинні повертати булевий результат. Не треба власноруч порівнювати між собою результати тестів у двох текстових файлах – це є неправильним підходом, тому що процес не є автоматизованим і витрачається значно більше часу;
- *Timely* – тести повинні бути написані перед production кодом, якщо ж навпаки, тоді можуть виникнути проблеми із тестуванням.

Плюси використання такого підходу:

- Можна використовувати в якості документації, іноді навіть краще звичайних прикладів, адже можна побачити усі потенційні помилки;
- Test-first – можливість спочатку написати тести, які будуть конкретизувати задачу, а потім вже код. Головною метою розробника є виконати задачу так, аби все тести пройшли успішно;
- Автоматичне тестування.

### 2.1.2 Бібліотека HUnit

У Haskell також є бібліотека для Unit-тестів – HUnit, назва якої пішла від цієї бібліотеки на мові програмування Java.

Основні типи та команди[5]:

- `Assertion` є синонімом типу `IO дії`. Він сигналізує про помилку, викликаючи функцію `assertFailure`;

```
type Assertion = IO ()
```

- Функція `assertBool` приймає стрічку для повідомлення при виникненні помилки та умову. В результаті функція стверджує, що вказана умова виконується;

```
assertBool :: String -> Bool -> Assertion
```

- Функція `assertEqual` приймає опис тесту у вигляді стрічки та два параметри, очікуваний та фактичний результат. Функція стверджує, що очікуваний результат еквівалентний до фактичного;

```
assertEqual :: (Eq a, Show a) => String -> a -> a -> Assertion
```

- Функція `assertFailure` приймає стрічку із помилкою та викликає виключну ситуацію. Може використовуватись на пряму, але зазвичай використовується у перевірочних функціях, наприклад `assertBool`, `assertEqual`;

```
assertFailure :: String -> Assertion
```

- Тип `Test` є рекурсивним та має три конструктори типу. `TestCase` є одним тест кейсом, а `TestList` – групою тестів. За допомогою конструктора типу `TestLabel` можна описати тест(и).

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

Фреймворк `Hspec` інтегрується з бібліотекою `HUnit` та має вбудовану функцію `fromHUnitTest` для конвертації `HUnit` тестів до `Hspec` тестів:

```
fromHUnitTest :: Test -> Spec
```



### 2.1.3. Приклад написання Unit-тестів з використанням HUnit

Щоб розпочати роботу, треба додати пакети HUnit, hspectest до залежностей та імпортувати модулі Test.Hspec.Contrib.HUnit, Test.HUnit у файлі Spec.hs. Необхідно протестувати функції evaluateCode та parseCode з файлу Lib.hs:

```
evaluateCode :: ProgramVariables -> Operator -> ProgramVariables
evaluateCode state (Assign variable expression) = updateValFromState state variable
  (evaluateExpression state expression)
evaluateCode state (Increment variable) = evaluateCode state
  (Assign variable (Op (Const value) Plus (Const (I 1))))
  where value = getValFromState state variable
evaluateCode state (If expression statement1 statement2) =
  if ((evaluateExpression state expression) == (B True))
  then evaluateCode state statement1 else evaluateCode state statement2
evaluateCode state (While expression smt) =
  if ((evaluateExpression state expression) == (B True))
  then evaluateCode (evaluateCode state smt) (While expression smt) else state
evaluateCode state (For smt1 expression smt2 smt3) =
  if ((evaluateExpression state1 expression) == (B True))
  then evaluateCode state2 (For (Block [] []) expression smt2 smt3) else state
  where
    state1 = evaluateCode state smt1
    state3 = evaluateCode state1 smt3
    state2 = evaluateCode state3 smt2
evaluateCode state (Block types stmts) =
  foldl1 (deleteVariable) (foldl1 (evaluateCode) state4 stmts) types
  where state4 = (calculateState types) ++ state
```

Функція evaluateCode приймає масив зі змінними та оператор, який може бути присвоєнням, інкрементом, умовним оператором If, циклом For/While або блоком. В результаті виконується оператор і повертається оновлений список змінних.

```

parseCode :: String -> Either ParseError Program
parseCode s = parse program "" s

```

Функція `parseCode` приймає код у вигляді стрічки і будує її представлення у вигляді AST(абстрактне синтаксичне дерево). AST для цієї задачі описується Haskell типами, що можна знайти у файлі `Lib.hs`.

Додаємо до файлу три Unit-тести із типом `Test`. Далі об'єднуємо їх у групу тестів:

```

forExample :: String
forExample =
    "{ int x; int y; int z; x := 3; y := 2; z:= 1;\
      \{ int i; for (i:=0; i<x; i++) z := z*x } \
    }"

ifExample :: String
ifExample = "{int y; int x; bool V; bool Z; y := 5; x := 6; V := true; Z := false; \
  \ if(V) x++ else y++ }"

test1 :: Test
test1 = TestCase (assertEqual "evaluateCode [...] (If ..)" (I 7)
  (getValFromState (evaluateCode [("y",(I 5)), ("x",(I 6)), ("V",(B True))]
    (If (Variable "V") (Increment "x") (Increment "y")) "x")))

test2 :: Test
test2 = TestCase (assertEqual "evaluateCode [...] (If ..) in text view" (I 7)
  (case parseCode ifExample of
    Left e  -> error $ show e
    Right r -> (getValFromState (evaluateProgram r) "x")))

test3 :: Test
test3 = TestCase (assertEqual "evaluateCode [...] (For ..) z in text view" (I 27)
  (case parseCode forExample of
    Left e  -> error $ show e
    Right r -> (getValFromState (evaluateProgram r) "z")))

tests :: Test
tests = TestList [TestLabel "Testing If case" test1,
  TestLabel "Testing If case in text view" test2,
  TestLabel "Testing For case in text view" test3]

```

У першому тесті розглядається випадок, коли функція `evaluateCode` приймає у якості оператора `If` блок, який в залежності від значення булевої змінної “`V`” інкрементує або змінну “`x`”, або змінну “`y`”. Після виконання умовного оператора `If` функція повертає оновлений список змінних, з якого треба отримати значення змінної “`x`” за допомогою функції `getValFromState`, що була розглянута у розділі 1.3.2. Отримане значення перевіряється із очікуванням за допомогою функції `assertEqual`.

У другому тесті відбувається синтаксичний аналіз стрічки коду `ifExample` із оператором `If`. Якщо стрічка коду немає синтаксичних помилок, то аналізатор будує тип `Operator`, який далі перевіряється аналогічно до першого тесту.

Третій тест відрізняється від другого лише тим, що відбувається синтаксичний аналіз стрічки коду `forExample` із циклом `For`.

Наступним кроком треба запустити тести та перевірити їх коректність. Додаємо тести до функції `main`:

```
main :: IO ()
main = hspec $ do
  describe "Lib.getValFromState" $ do
    it "returns value of the second variable in list" $ do
      (getValFromState [("y", (I 5)), ("x", (I 6)), ("V", (B True))] "x")
        `shouldBe` ((I 6) :: Value)
    it "returns value of the first variable in list" $ do
      (getValFromState [("y", (I 5)), ("x", (I 6)), ("V", (B True))] "y")
        `shouldSatisfy` (== (I 5))
  describe "Lib.updateValFromState" $ do
    it "" $ pending
  describe "HUnit tests casted to hspec" $ do
    fromHUnitTest tests
```

Та запускаємо процес тестування в консолі за допомогою команди `stack test`. В результаті бачимо, що всі тести виконались успішно:

```
Lib.getValFromState
  returns value of the second variable in list
  returns value of the first variable in list
Lib.updateValFromState
Main[166:7]
# PENDING: No reason given
HUnit tests casted to hspec
Testing If case
Testing If case in text view
Testing For case in text view

Finished in 0.0014 seconds
6 examples, 0 failures, 1 pending

cursova> Test suite cursova-test passed
Completed 2 action(s).
```

## 2.2 Property testing

### 2.2.1 Загальні відомості

Підхід Property testing, виходячи з назви, базується на властивостях. А саме, за допомогою функцій у Haskell, необхідно сформулювати умову, яка буде повертати булевий результат, де істина – це успішно виконаний тест, а хиба – помилка. За конвенцією функції із властивостями зберігають у собі частинку `property` або `prop`, аби було зручно орієнтуватися у коді.

У файлі `Lib.hs` знаходиться функція `calculate`, котра на вхід приймає два значення, де значення може бути або числовим із конструктором типу `I`, або булевим із конструктором типу `B`. Третім аргументом функція приймає оператор (`Plus`, `Minus`, `Times`, `Div`, `Gt`, `Ge`, `Lt`, `Le`, `Eql`, `And`, `Or`), далі підраховує результат і повертає його:

```

calculate :: Value -> Value -> Bop -> Value
calculate (I i1) (I i2) Plus = I (i1 + i2)
calculate (I i1) (I i2) Minus = I (i1 - i2)
calculate (I i1) (I i2) Times = I (i1 * i2)
calculate (I i1) (I i2) Division = I (i1 `div` i2)
calculate (I i1) (I i2) GreaterThan = B (i1>i2)
calculate (I i1) (I i2) GreaterThanEqual = B (i1>=i2)
calculate (I i1) (I i2) LessThan = B (i1<i2)
calculate (I i1) (I i2) LessThanEqual = B (i1<=i2)
calculate (I i1) (I i2) Equal = B (i1==i2)
calculate (B b1) (B b2) And = B (b1 && b2)
calculate (B b1) (B b2) Or = B (b1 || b2)
calculate _ _ _ = error "The expression is not correct!"

```

Щоб сформулювати властивість для функції `calculate`, треба у файлі `Spec.hs` написати функцію `calculateTest`, котра буде аналогічною до функції `calculate` і завжди буде повертати правильний результат:

```

calculateTest :: Value -> Value -> Bop -> Value
calculateTest (I i1) (I i2) o
  | o == Plus = I (i1+i2)
  | o == Minus = I (i1 - i2)
  | o == Times = I (i1*i2)
  | o == Division = I (i1 `div` i2)
  | o == GreaterThan = B (i1>i2)
  | o == GreaterThanEqual = B (i1>=i2)
  | o == LessThan = B (i1<i2)
  | o == LessThanEqual = B (i1<=i2)
  | o == Equal = B (i1 == i2)
  | otherwise = error "Such operation are not implemented for integer"
calculateTest (B b1) (B b2) o
  | o == And = B (b1 && b2)
  | o == Or = B (b1 || b2)
  | otherwise = error "Such operation are not implemented for boolean"
calculateTest _ _ _ = error "Not correct expression"

```

На початку, щоб зрозуміти цей підхід і правильно написати властивість, треба сформулювати умову у вигляді речення. На прикладі завдання роботи, сформульовано наступне:

- Виклик функції `calculateTest` повинен повертати такий же результат, що і `calculate`, де у якості `calculate` використовуємо функцію, що необхідно протестувати на коректність роботи, а `calculateTest` – функція, що буде повертати завжди правильний результат. Як бачимо вони досить примітивні, але навіть і тут можна помилитись, наприклад забути про ділення на нуль і т. д. Зручність полягає у тому, що тепер модифікуючи функцію `calculate` можна не хвилюватись за логічні помилки.

У Haskell дане речення буде виглядати наступним чином:

```
data Expression = Expression Value Value Bop deriving (Show)

calcProperty :: Expression -> Bool
calcProperty (Expression v1 v2 op) = calculateTest v1 v2 op == calculate v1 v2 op
```

Створимо новий тип `Expression`, який буде мати єдиний конструктор типу `Expression`, що приймає два значення і арифметичний/логічний оператор. Властивість `calcProperty` приймає на вхід вираз, передає значення виразу до функцій `calculate`, `calculateTest` та перевіряє чи результати цих функцій еквівалентні.

Щоб перевірити властивість, можна власноруч викликати функцію, підставляючи туди різні параметри. Мінусом такого підходу є відсутність

автоматичних тестів, необхідно самому придумувати їх. Для вирішення цієї проблеми придумали бібліотеки QuickCheck та SmallCheck.

### 2.2.2 Бібліотека QuickCheck

QuickCheck – це утиліта для автоматичного тестування властивостей в Haskell. Головною її особливістю є можливість задати обмеження для параметрів функцій, після чого бібліотека сама згенерує тести з випадковими значеннями, рахуючи обмеження програміста. QuickCheck базується на підході Property testing. Бібліотека запускає 100 (кількість можна регулювати) разів властивість із різними параметрами, при цьому не перериваючи свою роботу на першій помилці, а підраховуючи загальну кількість успіху та хиб.

Переваги:

- За допомогою властивостей можна сформулювати специфікацію та вимоги до коду;
- Бібліотека перевіряє програму на відповідність до специфікації за допомогою тестування;
- Є гарною документацією для нових програмістів, де можна знайти які властивості тестуються, а які – ні;
- Головною перевагою є те, що не треба писати самому велику кількість до прикладу Unit-тестів, бібліотека сама генерує багато тестів;
- Можливість за допомогою генераторів робити найрізноманітніші обмеження для об'єктів тестування.

Недоліки:

- Погана документація бібліотеки;

- Невелика кількість прикладів.

Щоб бібліотека змогла згенерувати дані для властивостей, усі типи повинні бути екземпляром класу `Arbitrary`[6]:

```
class Arbitrary a where
  arbitrary  :: Gen a
  shrink    :: a -> [a]
```

Клас надає два методи: `arbitrary` та `shrink`. Перший генерує випадкові значення за допомогою генератора `Gen` для типу `a`.

Другий метод поступово знижує складність генеруючих даних – це допомагає у випадках, коли на деяких даних властивість видає помилку і бібліотека намагається звужити обсяг даних, щоб зрозуміти через що саме виникає помилка. Але на конкретному прикладі ми розглянемо саме перший метод.

Базові типи за замовченням є екземплярами цього класу, але їх не так багато. Більший набір екземплярів можна знайти у пакеті `quickcheck-instances`. Щоб генерувати дані для власних типів, необхідно зробити їх екземплярами класу `Arbitrary` та за необхідності написати власний генератор `Gen`.

Фреймворк `Hspec` підтримує властивості із типом `Property`. Для перетворення до типу `Property` у модулі `Test.QuickCheck.Property` є метод `property` з класу `Testable`:



```
class Testable prop where
  property :: prop -> Property
```

За замовчуванням бібліотека тестує властивість 100 разів, але цей параметр можна змінити за допомогою функції `withMaxSuccess`, що приймає на вхід кількість разів прогону властивості, саму властивість та повертає тип `Property`:

```
withMaxSuccess :: Testable prop => Int -> prop -> Property
```

### 2.2.3 Приклад написання тестів з використанням бібліотеки QuickCheck

На початку необхідно додати пакети `QuickCheck`, `quickcheck-instances` до файлу із залежностями та імпортувати модуль `Test.QuickCheck` у файлі `Spec.hs`.

Властивість `calcProperty`, котра детально була описана у розділі 2.2.1, приймає тип `Expression`, який не є стандартним, тому треба зробити його екземпляром класу `Arbitrary`. Але виникає проблема в контролі сумісності типів операторів і значень. Якщо генерувати разом логічні та арифметичні значення та оператори, то в результаті велика вірогідність отримати приклад тесту із один значенням булевим, а іншим числовим, або два булевих значення із оператором додавання або множення.

Щоб вирішити цю проблему, треба написати два окремих генератори для арифметичних та логічних виразів, а потім у методі `arbitrary` випадковим чином обирати один із двох генераторів.

Додаємо наступний код до файлу `Spec.hs`:

```
instance Arbitrary Expression where
  arbitrary = oneof [genIntegerExp, genBoolExp]

genNotZeroInt :: Gen Int
genNotZeroInt = abs `fmap` (arbitrary :: Gen Int) `suchThat` (/= 0)

genIntegerExp :: Gen Expression
genIntegerExp = do
  i1 <- arbitrary :: Gen Int
  i2 <- genNotZeroInt
  op <- elements [Plus, Minus, Times, Division, GreaterThan,
    GreaterThanEqual, LessThan, LessThanEqual, Equal]
  return (Expression (I i1) (I i2) op)

genBoolExp :: Gen Expression
genBoolExp = do
  b1 <- arbitrary :: Gen Bool
  b2 <- arbitrary :: Gen Bool
  op <- elements [And, Or]
  return (Expression (B b1) (B b2) op)
```

`genIntegerExp` генерує значення для арифметичних виразів. За допомогою вбудованого генератора для чисел генеруємо перше значення, реалізуємо додатковий генератор `genNotZeroInt` для другого значення та обираємо навмання один з арифметичних операторів. Додатковий генератор потрібен для генерування будь-якого числа, окрім нуля, щоб запобігти виключної ситуації з діленням на нуль.

`genBoolExpr` генерує значення для логічних виразів. За допомогою вбудованого генератора для булевого типу генеруємо перше та друге значення, для третього значення обираємо навмання один з логічних операторів.

Розглянемо детальніше у табличному вигляді, які вирази генерують генератори:

Генератор	Перший операнд	Другий операнд	Оператор
Для арифметичних значень	Будь-яке число	Будь-яке число, окрім 0	+, -, *, /, >, >=, <, <=, ==
Для логічних значень	True, False	True, False	&&,

Таблиця 1 Опис випадкових генераторів для логічних та арифметичних виразів

Додаємо до функції `main` у файлі `Spec.hs` блок тестів, пов'язаний з бібліотекою `QuickCheck` та задаємо 200 тестів:

```
describe "Lib.calculate" $ do
  it "quick property testing with random generated values" $ do
    withMaxSuccess 200 calcProperty
```

Запускаємо тести у консолі:

```

Lib.getValFromState
  returns value of the second variable in list
  returns value of the first variable in list
Lib.updateValFromState
Main[166:7]
  # PENDING: No reason given
HUnit tests casted to hspec
Testing If case
Testing If case in text view
Testing For case in text view
Lib.calculate
  quick property testing with random generated values
  +++ OK, passed 200 tests.

Finished in 0.0043 seconds
7 examples, 0 failures, 1 pending

cursova> Test suite cursova-test passed
Completed 2 action(s).

```

Як бачимо, всі тести виконались успішно. Можна одразу оцінити зручність цього підходу і гнучкість. Він економить багато часу, який можливо було витратити на написання 200 Unit-тестів. Також бібліотека на Haskell дуже зручна та містить увесь необхідний функціонал.

## 2.2.4 Бібліотека SmallCheck

SmallCheck – це бібліотека, яка дозволяє виконати для властивостей тести, використовуючи додатковий параметр глибини. Концепція схожа на QuickCheck, тому що самі тести генеруються автоматично, а робота програміста обмежити дані, що приймає на вході властивість. Різниця між двома вище згаданими бібліотеками полягає у тому, що QuickCheck випадково генерує значення із обмеженого набору даних, у той час коли SmallCheck перевіряє усі кінцеві значення до певної глибини.

У SmallCheck бібліотеці аналогією до Arbitrary класу є Serial клас[7]:

```
type Depth = Int
class Monad m => Serial m a where
  series :: Series m a
```

Метод `series` повертає тип `Series`, який є дією, котра перераховує значення певного типу до певної глибини.

За допомогою функції `generate` можна перетворити функцію, що приймає параметр глибини та повертає масив значень, у тип `Series`:

```
generate :: (Depth -> [a]) -> Series m a
```

Щоб протестувати властивість за допомогою фреймворку Hspec, у модулі `Test.Hspec.SmallCheck`[8] знаходиться функція `property` для конвертування у тип `Property IO`:

```
property :: Testable IO a => a -> Property IO
```

### 2.2.5 Приклад написання тестів з використанням бібліотеки SmallCheck

На початку необхідно додати бібліотеку `smallcheck` та `hspec-smallcheck` до файлу із залежностями та імпортувати модулі `Test.SmallCheck.Series`, `Test.Hspec.SmallCheck` у файлі `Spec.hs`.

У випадку із `QuickCheck` бібліотекою, аби уникнути проблем із сумісністю арифметичних та логічних значень, необхідно було створити новий тип `Expression` та створювати окремі генератори для кожного з виразів, як описано у розділі 2.2.3. З бібліотекою `SmallCheck` можна уникнути цих зайвих дій та використати параметр глибини:

```
operators :: [Bop]
operators = [Plus, Minus, Times, Division, GreaterThan,
            GreaterThanEqual, LessThan, LessThanEqual, Equal, And, Or]

ops 1 = take 9 operators --generate only operators for integer
ops _ = take 2 $ drop 9 operators

vals 1 = map (\x -> (I x)) [1..100]
vals _ = map (\x -> (B x)) [True, False]

instance Monad m => Serial m Bop where
    series = Test.SmallCheck.Series.generate ops

instance Monad m => Serial m Value where
    series = Test.SmallCheck.Series.generate vals
```

Створюємо функцію `ops`, яка в залежності від числа повертає різний список операторів. Якщо параметр глибини дорівнює 1, то повертається

список арифметичних операторів, якщо будь-який інший – то повертається список логічних операторів. За схожою схемою працює функція `vals`, але повертає замість списку операторів список значень. Якщо параметр глибини дорівнює 1, то повертається список чисел від 1 до 100, якщо будь-який інший – то повертається список із двома булевими значеннями істини і хиби.

Щоб бібліотека згенерувала дані для властивостей, типи повинні бути екземплярами класу `Serial`. Тому робимо типи `Bool` та `Value` екземплярами класу `Serial` та реалізуємо метод `series`. Щоб перетворити масив значень у `Series` тип, використовуємо функцію `generate`, котра приймає функцію `ops/vals`.

Розглянемо, що відбувається під час генерування даних із різним параметром глибини у табличному вигляді:

Глибина	Оператори	Операнди
1	<code>+, -, *, /, &gt;, &gt;=, &lt;, &lt;=, ==</code>	Від 1 до 100
2 та більше	<code>&amp;&amp;,   </code>	<code>True, False</code>

*Таблиця 2 Огляд згенерованих даних, в залежності від параметру глибини*

Додаємо до функції `main` у файлі `Spec.hs` блок тестів, пов'язаний з бібліотекою `SmallCheck`:

```
describe "Lib.calculate" $ do
  it "small property testing" $ do
    Test.Hspec.SmallCheck.property $ \v1 v2 op -> calculateTest
      (v1 :: Value) (v2 :: Value) (op :: Bop) == calculate v1 v2 op
```

Запускаємо тести у консолі, щоб переконатись, що тести виконались успішно:

```
Lib.getValFromState
  returns value of the second variable in list
  returns value of the first variable in list
Lib.updateValFromState
  Main[145:7]
  # PENDING: No reason given
HUnit tests casted to hspec
  Testing If case
  Testing If case in text view
  Testing For case in text view
Lib.calculate
  quick property testing with random generated values
  +++ OK, passed 200 tests.
Lib.calculate
  small property testing
Finished in 0.0053 seconds
8 examples, 0 failures, 1 pending
cursova> Test suite cursova-test passed
Completed 2 action(s).
```

## 2.3 Mocking

### 2.3.1 Загальні відомості

Інколи під час написання тестів необхідно використовувати об'єкти, які тягнуть за собою багато залежностей. Аби уникнути таких проблем було винайдено підхід створити інший об'єкт, який буде симулювати поведінку реального. Це називається *mocking*, тобто імітацією. Зазвичай його використовуються разом із Unit-тестами. Підхід дуже популярний не тільки у



Haskell, а й в інших мовах програмування та зазвичай дуже полегшує роботу із імітацією бази даних, адже ці дії важко простежити і неможливо передбачити результат. У цій роботі розглядається приклад із імітацією зчитування тексту з клавіатури .

Переглянувши документацію було знайдено декілька способів як це можливо зробити. Один із них використання бібліотеки `moskazo`, а другий за допомогою вбудованих монад. Розглянемо більш детально другий варіант, поглибившись у специфіку мови.

Haskell – це мова програмування, у якій усі дані не змінюються. Це означає, що функції не змінюють значення комірок пам’яті, вони лише приймають значення та повертають результат. Усі функції, які дотримуються цих правил, називаються “чистими”, тому що за будь-яких умов повертають прогнозований результат. Але є виняток – монада `IO`, через те що її поведінка і результат залежить від користувача.

Базові операції для роботи з монадою `IO`[9]:

- `getChar` – функція, що повертає `IO` дію, котра зчитує символ з вхідного потоку;
- `putChar` – функція, що приймає символ та повертає `IO` дію, котра записує символ у вихідний потік;
- `getLine` – функція, що повертає `IO` дію, котра зчитує стрічку з вхідного потоку;
- `putStr` – функція, що приймає стрічку та повертає `IO` дію, котра записує стрічку у вихідний потік;
- `putStrLn` – функція, що приймає стрічку та повертає `IO` дію, котра записує стрічку у вихідний потік та стрибає на новий рядок.

### 2.3.2 Приклад написання тестів

Головною функцією програми, котра згадувалась у розділі 1.1.2, є зчитування назви файлу із кодом, який надалі аналізується і інтерпретується. Проблемою стало тестування такої функції, адже зчитування з клавіатури повертає непередбачуваний результат. Щоб її вирішити, треба написати власну монаду для тестування, яка імітує поведінку, тобто функції для роботи з монадою `IO`. Створюємо новий файл `MockSample.hs` у каталозі `src`. У новому файлі створюємо власний клас `TestMonadIO` із обмеженням, що його екземплярами можуть бути лише ті типи, що є монадою. Для тестування необхідна лише функція зчитування стрічки, тому клас містить один метод `testGetLine`:

```
class Monad m => TestMonadIO m where
    testGetLine :: m String

instance TestMonadIO IO where
    testGetLine = Prelude.getLine
```

Робимо монаду `IO` екземпляром створеного класу. Для реалізації методу `testGetLine` використовуємо функцію `getLine` з модулю `Prelude`.

Щоб зберігати завчасно підготовлені дані, які імітують дані, зчитані з клавіатури, використовуємо монаду `State` з модулю `Control.Monad.State`. Додаємо новий тип `TestMonadState`, у якому будуть зберігатись назви файлів із кодом:

```
data TestMonadState = TestMonadState
{
    output :: [String]
}
deriving (Show, Eq)
```

Монада State має наступний вигляд:

```
newtype State s a = State { runState :: s -> (a,s) }
```

де  $s$  - це стан, для якого щойно було створено тип `TestMonadState`, а  $a$  – це тип, який повертає функція `runState`, у конкретному випадку це стрічка, тобто назва файлу.

Далі монаду `State TestMonadState` необхідно зробити екземпляром `TestMonadIO`, як було зроблено із `IO` монадою. Але якщо для монади `IO` викликалась функція зчитування з клавіатури, визначена в модулі `Prelude`, то зараз необхідно написати свою реалізацію:

```
instance TestMonadIO (State TestMonadState) where
  testGetLine = do
    st <- get
    case output st of
      [] -> error "Error"
      (x:xs) -> do
        let newState = st { output = xs }
        put newState
        return x
```

Спочатку метод дістає поточний стан, витягує з нього масив із назвами файлів `output`. Далі оновлює стан, присвоївши полю `output` всі елементи зі списку, окрім першого. В результаті метод повертає перший елемент із назвою файлу.

В файлі `Lib.hs` створюємо функцію, котра буде повертати `m String`, де `m` повинна бути екземпляром класу `TestMonadIO`:

```
getName :: TestMonadIO m => m String
getName = testGetLine
```


Тепер, якщо в функції `main` у файлі `Main.hs` змінити виклик функції `getLine` на новостворену `getName`, то вона буде працювати, як зчитування стрічки з клавіатури:

```
module Main where
import Lib

main :: IO ()
main = do
    name <- getLine
    x <- parseFile name
    putStrLn $ show $ x
```

```
module Main where
import Lib

main :: IO ()
main = do
    name <- getName
    x <- parseFile name
    putStrLn $ show $ x
```



А якщо функцію `getName` викликати у тесті, то вона буде повертати завжди фіксоване значення.

Тепер повернемося до написання тесту. У файлі `Spec.hs` імпортуємо модуль `MockSample` та додаємо до функції `main` тест:

```
describe "getName+parseFile" $ do
    it "mocking getLine method" $ do
        values <- parseFile res
        values `shouldBe` ([("a", I 317), ("b", I 17)])
    where
        (res, _) = runState (getName) (TestMonadState ["power.txt"])
```

За допомогою функції `runState` отримуємо з поточного стану назву файлу, передаємо її у функцію `parseFile`, що зчитує код з цього файлу, виконує його і повертає значення, яке далі порівнюється із очікуваним результатом. За допомогою такого тесту ми можемо впевнитись, що

програма працює коректно, адже якщо хоча б один процес з усіх вищесказаних перестане працювати, тест буде видавати помилку.

В консолі виконуємо команду `stack test` та перевіряємо чи успішно виконались усі тести:

```
Lib.getValFromState
  returns value of the second variable in list
  returns value of the first variable in list
Lib.updateValFromState
Main[145:7]
  # PENDING: No reason given
HUnit tests casted to hspect
Testing If case
Testing If case in text view
Testing For case in text view
Lib.calculate
  quick property testing with random generated values
  +++ OK, passed 200 tests.
Lib.calculate
  small property testing
getName+parseFile
  mocking getLine method

Finished in 0.0076 seconds
9 examples, 0 failures, 1 pending

cursova> Test suite cursova-test passed
Completed 2 action(s).
```

Отже, підсумовуючи, на прикладі видно, що даний підхід дуже зручний, не займає багато часу для написання і доволі простий для розуміння. Haskell дає можливість його використовувати через різні бібліотеки і не обмежує в реалізації, тому тести досить гнучкі.

## Висновок

Ознайомившись із багатьма бібліотеками для тестування у Haskell, можна зробити висновок, що процес тестування на перший погляд може здаватися непростим, але насправді це не так.

За допомогою фреймворку Hspec можна швидко і зручно організовувати тести та структурувати їх. Фреймворк також пропонує загальний функціонал для тестування. Великим плюсом цього фреймворку є інтеграція з іншими бібліотеками для тестування.

За допомогою бібліотеки HUnit можна писати загальновідомі Unit-тести. Тести автоматизовані та виглядають досить лаконічно.

За допомогою бібліотеки QuickCheck можна випадковим чином генерувати тисячі тестів за декілька секунд, що економить багато часу. Все, що необхідно – написати обмеження для даних. Якщо підхід із випадковим генеруванням даних не підходить, можна скористатись бібліотекою SmallCheck, котра перевірить усі значення до певної глибини.

Як і в інших мовах програмування, у Haskell можна симулювати поведінку різних об'єктів за допомогою підходу Mocking. Даний підхід дозволяє підмінити речі, які не можна контролювати.

Отже, підбиваючи підсумки, можна сказати, що всі розглянуті концепції тестування є зручними та кожна з них позбавляє багатьох турбот. Тестування полегшує життя та вирішує безліч проблем.

## Список використаної літератури

1. Уилл К. Програмируй на Haskell. ДМК Пресс, М., 2019
2. stack.yaml vs cabal package files - The Haskell Tool Stack[Електронний ресурс] – Режим доступу до ресурсу:  
[https://docs.haskellstack.org/en/stable/stack\\_yaml\\_vs\\_cabal\\_package\\_file/](https://docs.haskellstack.org/en/stable/stack_yaml_vs_cabal_package_file/)
3. Test.Hspec[Електронний ресурс] – Режим доступу до ресурсу:  
<https://hackage.haskell.org/package/hspec-2.7.1/docs/Test-Hspec.html>
4. Мартин, Роберт К. Чистый код : создание, анализ и рефакторинг / Роберт Мартин ; [пер. с англ. Е. Матвеев]. - Санкт-Петербург : Питер, 2016. - 464 с. : ил., портр., табл. - Містить покажчик.
5. HUnit: A unit testing framework for Haskell[Електронний ресурс] – Режим доступу до ресурсу: <https://hackage.haskell.org/package/HUnit>
6. Test.QuickCheck[Електронний ресурс] – Режим доступу до ресурсу:  
<https://hackage.haskell.org/package/QuickCheck-2.14/docs/Test-QuickCheck.html>
7. Test.SmallCheck.Series[Електронний ресурс] – Режим доступу до ресурсу: <http://hackage.haskell.org/package/smallcheck-1.1.5/docs/Test-SmallCheck-Series.html>
8. Test.Hspec.SmallCheck [Електронний ресурс] – Режим доступу до ресурсу: <https://hackage.haskell.org/package/hspec-smallcheck-0.5.2/docs/Test-Hspec-SmallCheck.html>
9. Input and Output - Learn You a Haskell for Great Good![Електронний ресурс] – Режим доступу до ресурсу: <http://learnyouahaskell.com/input-and-output>